Chandra X-Ray Center
MEMORANDUM
March 4, 2003

To: SDS, DS
From: Douglas Burke, John E. Davis, John Houck, Michael Noble, and Randall Smith
Subject: Recommendations for S-Lang Module Development and Distribution

1 Introduction

The primary motivation for embedding an interpreter such as S-Lang into CIAO is to provide the end user with the means to extend the software in ways that were not anticipated by the designers of the software. Modules extend this capability to libraries by providing the end-user with access to the libraries via the scripting language.

A module is defined to be a shared object, usually written in C, that is dynamically linked into a S-Lang application during runtime via the S-Lang `import` statement. Examples of CXC developed modules include the fits, cxcparam, and xpa modules, with more on the horizon. Before these modules are made available to the public, we feel that it is necessary to adopt some minimal standards to which the modules should conform. A module that complies with the standards set forth in this memo should work well in any application that supports dynamically loaded slang modules.

We also address the issue of how the modules are to be disseminated to the public. It is felt that providing CXC developed modules to the users solely as part of a CIAO distribution would do a disservice to the community and would be an impediment to module development. Requiring full buy-in from the outset may discourage those who would prefer to test the paradigm by using a module-based tool. There is reflexive wariness from the user community, especially scientists, about any new language (thus FORTRAN's enduring popularity) and so minimal buy-in and maximal functionality must be stressed for this approach to succeed. Therefore, we argue that modules be released to the public separately on a module by module basis and, optionally, as part of a module package. Of course these additional avenues of distribution does not preclude modules from being distributed with CIAO. To this end, will propose that a S-Lang modules webpage will be created. Among other things, the page will contain links to existing modules, developer tools, and documentation.

Although a module may be distributed with a one or more S-Lang scripts that provide a higher level interface to the module intrinsics, it is important to understand that a set of interrelated S-Lang scripts do not constitute a module. Rather, we shall speak of the interrelated S-Lang scripts as a *package*. A recommended set of guidelines applicable to packages will be presented in a future memo.

2 Module Requirements

We feel that a module must meet the following set of requirements:

- A module must be importable into any slang application that supports dynamic loading through the import statement.
- The module must support loading into either the "Global" namespace (the default) or a user-specified namespace. For example, if the user loads a module "foo" via

```
import ("foo", "bar");
```

then any symbols created by the model must be loaded into the namespace "bar".

- The module must provide a version number. A proposal for module versioning is presented below.
- If the library wrapped by the module provides a version number, then the module must provide a symbol representing that version number. This number should be not be confused with the version number of the module itself. For example, the version number of the fits module is 1.1.0, whereas the cfitsio library that it wraps has a version number of 2.440.
- If a module is distributed with one or more associated .sl files, then the code in those files must be generic enough to work with slsh. This requirement permits easy testing of the module and provides a high degree of certainty that that the module will work in other applications (sherpa, chips, isis, etc).
- A module shall not load other .sl files or other modules. If use of the module is facilitated by code in a higher-level .sl file, then let that file load the module— not the other way around.

We will be providing a nominal module compliance test script to aid the developer in meeting these requirements.

3 Version Numbers

We propose that each module has a version number that allows for a major version number, a minor version number, and a patch level.

Increments in the major version number should occur only in new releases of the module where changes were made that break backward compatibility with previous versions. If the internals of the module were changed, but the scripting interface was kept the same, then the major version should be kept the same, and the minor version number incremented. Bug-fix releases should keep the same major and minor versions but use an increased patch level version number.

The easiest way to handle these three numbers is to encode the numbers into an integer as follows:

```
Version = Major*10000 + Minor*100 + Patchlevel
```

This scheme has a clear advantage in that it promotes convenient arithmetic comparisons and can be easy to verify in compliance scripts. We also recommend that the version number be made available as a string in the form "Major.Minor.Patchlevel". These module should make these values available through variables with names of the form:

```
_<module>_version
_<module>_version_string
```

For example, a module "foo" with a major version of 1, minor version of 3, and patch level 2 would create the two variables:

```
_foo_version        = 10302
_foo_version_string = "1.3.2"
```

Finally, it may be useful to adopt the common convention that releases with an odd value for the minor version number are to be regarded as development or test releases, and that those with even minor version numbers are considered to be stable.

4 Packaging and Distribution

We recommend that each module be made available to users in source code form as a compressed tar file. The distribution should contain an autoconf generated configure script that allows the user to easily compile and install the module, e.g.,

```
./configure
make
make install
```

Template files will be made available to module developers who are not familiar with autoconf.

The source code to the library being wrapped by the module should not be part of the distribution. During the module build process, the module configure script will check that the proper libraries are available on the target system, and fail when they are not found. It is up to the user to ensure that the appropriate libraries are installed.

We strongly recommend that future distributions of CIAO binaries be shipped with the so-called header files associated with the libraries in the distribution[1]. This would allow the end user to not only compile the modules that depend upon CIAO libraries, but to develop new CIAO based modules.

The current mechanism for compiling against a CIAO library in the public release includes downloading both the CIAO dependencies (OTS) and CIAO source, and building *all* of both. This is the only way users can be assured of having *all* of the necessary CIAO header files collected within a single directory. By bundling the header files within the CIAO binary distribution we permit users to avoid this time-consuming and error-prone process.

Finally, note that shipping header files with binary distributions is a longstanding practice within the software community. Developers wishing to write an X11 client or compile a C program, for example, do not need *the entire X11 or GCC source* to do so, but rather only the respective header files and binaries.

It needs to be emphasized that the distribution of modules in source code form does not in any way preclude the distribution of pre-compiled modules, e.g. within CIAO. Rather, we feel that providing independent source code distributions would do more to encourage the use of modules, spur further development of them, and increase the rate at which enhancements flow back to users (who would need not wait 6-12 months for enhancements to appear in the next public CIAO release). As a point of reference we note that the widespread use of the CFITSIO library is in part due to its adoption of a similar release strategy.

5 Documentation

It is important that each intrinsic function provided by the module be documented, and that such be included with the module distribution. In keeping with all other intrinsic functions provided by the slang library, we recommend that the documentation be provided in text-macro format. We will provide a script that converts documentation from this format to the standard CIAO XML format.

We note that if the library being wrapped by a module already provides copious documentation it may not be desirable or practical to replicate such in the module distribution. The existing fitsio and SLgtk modules, and forthcoming GSL module, are examples of such. It is conceivable that in these cases the module documentation will be relatively small, consisting perhaps of

- an indication of *how* the module wraps the library.
- an identification of divergence points between the module API and library API.
- references to the existing library documentation.

6 Summary

In this memo, several recommendations regarding modules were made:

- A module should conform to the set of requirements outlined in sections 2 and 3.

[1] We are pleased to note that as of CIAO 2.3, the binary distribution does include the CIAO header files. However, a few of the header files are missing from the distribution making it impossible to interface to many of the libraries. We hope that this oversight will be remedied for the CIAO 3.0 release.

- Modules should be distributed separately from CIAO on a module-by-module basis. This would be facilitated through the creation of a S-Lang modules webpage.
- To spur further development of modules based upon CIAO libraries, thereby potentially increasing its user base, the header files associated with the CIAO libraries should be distributed as an integral part of the CIAO binary distribution.

In the next few weeks, we will make available a "template" module distribution that should expedite the creation of modules that conform to the recommendations made here. Also, a forthcoming memo will present a set of guidelines for dealing with issues such as "namespaces" and so on, that could arise when using multiple S-Lang packages.